MATEUS FELIPE DE CÁSSIO FERREIRA - GRR20176123

RELATRÓRIO DA AVALIAÇÃO DE DESEMPENHO DE ALGORITMOS OTIMIZADOS PARA MULTIPLICAÇÃO DE MATRIZES POR VETORES E RESOLUÇÃO DE SISTEMAS LINEARES

(*versão pré-defesa, compilada em 17 de junho de 2019*)

Trabalho apresentado como requisito parcial à conclusão da disciplina CI164 - Introdução à Computação Científica no Curso de Bacharelado em Informática Biomédica, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Informática Biomédica*.

Orientador: Daniel Weingaertner.

CURITIBA PR

2019

RESUMO

O presente trabalho busca apresentar a metodologia utilizada na avaliação de *performance* de dois conjuntos de algoritmos utilizados frequentemente no ambiente da computação científica. É apresentado, ainda, a sequência de testes realizados para tornar o experimento reprodutível em outros contextos. Para que esse experimento fosse realizado, foi necessária a utilização ferramentas de avaliação de performance, como o LIKWID e uma função para medição do tempo de execução. Foi constatado, pela análise comparativa entre as duas versões de algoritmos (com e sem otimização) que o grupo de funções matmult soube aproveitar melhor as otimizações feitas, utilizando o conjunto de registradores que suportam as instruções AVX. O outro grupo, SistemasLineares, por sua vez, obteve algumas melhores com relação a um ganho na quantidade de operações em ponto flutuante de precisão dupla realizados pelo processador, porém não conseguiu utilizar os registradores de forma a suportar as instruções AVX. Palavras-chave:

Avaliação de *performance* 1. Computação Científica 2. Otimização de código 3.

ABSTRACT

The present work seeks to present the methodology used in the evaluation of performance of two sets of algorithms frequently used in the scientific computing environment. It is also presented the sequence of tests performed to make the experiment reproducible in other contexts. For this experiment to be performed, it was necessary to use performance evaluation tools, such as LIKWID and a function to measure execution time. It was verified by the comparative analysis between the two versions of algorithms (with and without optimization) that the matmult function group was able to take better advantage of the optimizations made using the set of registers that support the AVX instructions. The other group, SistemasLineares, on the other hand, obtained some better gains in the amount of double-precision floating point operations performed by the processor, but it was not able to use the registers in order to support the AVX instructions.
Keywords: Performance Evaluation 1. Scientific Computing 2. Code Optimization 3.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

A otimização de código no universo da computação é um campo de estudos envolvendo diversas pesquisas e técnicas utilizadas para melhora a *performance* de um código em uma aplicação. Essa campo é de fundamental importância em sistemas embarcados, por exemplo, em que uma das principais características é a limitação de recursos computacionais, que devem ser bem utilizados.

A otimização de código está intimamente relacionada com a maneira em que é realizada a "transformação", ou compilação, do código-fonte no arquivo binário que será carregado para a memória do dispositivo Prado (2017). O compilador é a principal ferramenta para a geração de código eficiente, uma vez que ele realiza diversas transformações no código de alto nível escrito para gerar o melhor código possível para uma dada arquitetura. No entanto, essa ferramente não exime a responsabilidade do programador de escrever um código que seja facilmente otimizado pelo compilador.

Na área da computação científica, a otimização de código é uma ferramenta poderosa para aproveitar a arquitetura da máquina e os conjuntos de instruções mais utilizados para a geração de um código eficiente, uma vez que os problemas nessa área demandam um alto volume de processamento de dados. Assim, esses sistemas estão altamente susceptíveis a custos operacionais e alto consumo de energia.

A otimização de código permite, basicamente, a construção de programas mais rápidos, o que permite o uso de processadores mais lentos, com menor custo e menor consumo de energia; e programas mais curtos, com uma utilização menor da memória, logo menores custos e menor consumo de energia.

As áreas da otimização estão representadas na Figura 1.1 apresenta as principais áreas de num processo de otimização de código. Neste relatório, o objetivo é concentrar os esforços nas áreas da otimização escalar, vetorização e memória.

Dessa forma, foi determinado que quatro algoritmos deveriam ser otimizados. São eles:

- `multMatRowVet`: método de multiplicação de matrizes por vetores utilizando o acesso em *row major order* à matriz;
- `multMatColVet`: método de multiplicação de matrizes por vetores utilizando o acesso em *column major order* à matriz;
- `gaussJacobi`: método iterativo de resolução de sistemas lineares;
- `gaussSeidel`: método iterativo de resolução de sistemas lineares utilizando o resultado da iteração anterior para a construção da nova iteração;

Figura 1.1: Áreas da otimização de código.

O segundo capítulo apresenta as condições em que foram executados os testes de desempenho.

O terceiro capítulo apresenta a metodologia utilizada nos testes realizados, abordando a maneira com que as ferramentas foram utilizadas para a medição, bem como apresentando a forma correta a ser seguida para que se possa reproduzir os testes realizados utilizando o *script* desenvolvido.

Por fim, quarto capítulo apresenta os resultados obtidos nos testes realizados, exibindo os gráficos de performance dos algoritmos que foram gerados pelo *script*, com sua respectiva descrição, e analisando o desempenho de cada algoritmo em função dos parâmetros previamente estabelecidos. Ainda, é feito um comparativo com o ganho de *performance* das funções otimizadas em relação às antigas funções anteriormente implementadas.

2 DESENVOLVIMENTO

A fim de otimizar um algoritmo, previamente deve-se estabelecer parâmetros para definir a região do código em que o acesso à memória, bem como o processamento de dados é mais intenso. Este capítulo apresentará a técnicas de otimização usadas nos algoritmos.

2.1 USO DO *LOOP UNROLLING*

Essa técnica é utilizada para otimizar a velocidade de execução de um programa reduzindo, ou eliminando, instruções que controlam o *loop*, como testes de aritmética de ponteiros e o fim do *loop* a cada iteração. Esses *loops* são reescritos como uma sequência repetitiva de instruções independentes e semelhantes. A Figura 2.1 apresenta a aplicação dessa técnica no algoritmo do `gaussJacobi`.

2.2 USO DO *LOOP BLOCKING*

Essa técnica é utilizada para otimizar a reutilização de dados locais. Ela é essencial em *loops* aninhados que manipulam matrizes e que são muito grandes para caber na *cache*. Essa técnica permite, com isso, utilizar as linhas de cache que são trazidas pelo processador de forma que ele realize o maior número de cálculos em dados que já residem na *cache*. Assim, essa técnica permite a reutilização de dados tanto temporalmente, quanto espacialmente. A Figura 2.2 apresenta a aplicação dessa técnica no algoritmo do `multMatRowVet`.

```
while (*iter < MAXIT)
{
    (*iter)++;
    iFator = 0;
    idx = 0;
    for (i = 0; i < n; ++i) //trata da "linha"
    {
        R0 = R1 = R2 = R3 = R4 = R5 = R6 = R7 = soma = 0.0;
        posicao = (i * n);
        for (j = 0; j < n / unroll_factor; ++j) //trata da "coluna"
        {
            R0 += (SLc->A[posicao + j] * SLc->x[j]);
            R1 += (SLc->A[posicao + j+1] * SLc->x[j+1]);
            R2 += (SLc->A[posicao + j+2] * SLc->x[j+2]);
            R3 += (SLc->A[posicao + j+3] * SLc->x[j+3]);
            R4 += (SLc->A[posicao + j+4] * SLc->x[j+4]);
            R5 += (SLc->A[posicao + j+5] * SLc->x[j+5]);
            R6 += (SLc->A[posicao + j+6] * SLc->x[j+6]);
            R7 += (SLc->A[posicao + j+7] * SLc->x[j+7]);
        }
        //remainder
        for (j = n - (n / unroll_factor); j < n; ++j)
        {
            soma += (SLc->A[posicao+j] * SLc->x[j]);
        }

        RPP1 = R0 + R4;
        RPP2 = R1 + R5;
        RPP3 = R2 + R6;
        RPP4 = R3 + R7;

        RP1 = RPP1 + RPP2;
        RP2 = RPP3 + RPP4;
```

Figura 2.1: *Loop unrolling* no algoritmo de gaussJacobi

```
for (ii = 0; ii < n / b; ++ii)
{
  istart = ii * b;
  iend = istart + b;

  for (jj = 0; jj < n / b; ++jj)
  {
    jstart = jj * b;
    jend = jstart + b;
    j = jstart;
    ind = 0;

    for (i = istart; i < iend; ++i)
    {
      R0 = mat[(m * i) + j] * v [j];
      R1 = mat[(m * i) + j+1] * v [j+1];
      R2 = mat[(m * i) + j+2] * v [j+2];
      R3 = mat[(m * i) + j+3] * v [j+3];
      R4 = mat[(m * i) + j+4] * v [j+4];
      R5 = mat[(m * i) + j+5] * v [j+5];
      R6 = mat[(m * i) + j+6] * v [j+6];
      R7 = mat[(m * i) + j+7] * v [j+7];

      RPP1 = R0 + R4;
      RPP2 = R1 + R5;
      RPP3 = R2 + R6;
      RPP4 = R3 + R7;

      RP1 = RPP1 + RPP2;
      RP2 = RPP3 + RPP4;

      res [j + ind] = RP1 + RP2;
      ind++;
    }
```

Figura 2.2: *Loop blocking* no algoritmo do multMatRowVet

3 METODOLOGIA

A primeira parte da metodologia consistiu em determinar o conjunto de algoritmos que serão testados durante as avaliações de performance. O primeiro deles está contido no programa `matmult`, que basicamente efetua diferentes maneiras de multiplicação de uma matriz por um vetor. O segundo engloba algoritmos para calcular a resolução de Sistemas de Equações Lineares e está no programa `SistemasLineares`.

Já a segunda parte, baseou-se em separar os testes realizados em dois grupos:

- o teste de tempo de execução, utilizando a função timestamp ( );
- o grupo de testes que não dependiam de tempo (L3, CACHEL2 e FLOPS_DP), utilizando o LIKWID.

Essa separação foi importante para evitar que a medição do tempo de execução pudesse sofrer distorções por conta dos outros testes realizados nos algoritmos. Sendo assim, para os dois programas que serão testados existem marcadores específicos para determinar o tempo de execução de cada função. A Figura 3.1 mostra os marcadores utilizados no programa `matmult` para o cálculo do tempo. Já a Figura 3.2 mostra os marcadores utilizados no programa `SistemasLineares`.

```
printf("%d  ", n);

tempo = timestamp();
multMatRowVetOPT (mRow, vet, n, n, resRow);
tempo  = timestamp() - tempo;
printf("%.8g  ",tempo);


tempo = timestamp();
multMatColVetOPT (mCol, vet, n, n, resCol);
tempo  = timestamp() - tempo;
printf("%.8g  \n",tempo);


//norma = normaMax(resRow, resPtr, n);

//norma = normaEucl(resCol, n);
```

Figura 3.1: Marcadores de tempo utilizados no programa matmult.

Para o outro agrupamento de testes, os grupos L3, L2CACHE e FLOPS_DP utilizaram a mesma ferramenta de marcador disponível no LIKWID que permite delimitar uma região de interesse no algoritmo a fim de avaliar o desempenho daquela região para diferentes parâmetros.

```
printf("%d  ", n);

tempoL = timestamp();
verificador = gaussJacobiOPT (SL, EPS, &normaL2, &iter);
tempoL  = timestamp() - tempoL;
printf("%.8g  ",tempoL);

tempoL = timestamp();
verificador = gaussSeidelOPT (SL, EPS, &normaL2, &iter);
tempoL  = timestamp() - tempoL;
printf("%.8g  \n",tempoL);

//medição da média do tempo total da normaL2Residuo
//printf("%.8g  \n",tempoNormaL2/conta_entrada);
```

Figura 3.2: Marcadores de tempo utilizados no programa SistemasLineares.

A Figura 3.3 mostra os marcadores utilizados no programa `matmult` e a Figura 3.4 mostra os marcadores utilizados no programa `SistemasLineares`.

```
LIKWID_MARKER_INIT;

LIKWID_MARKER_START("multMatRowVetOPT");
multMatRowVetOPT (mRow, vet, n, n, resRow);
LIKWID_MARKER_STOP("multMatRowVetOPT");

LIKWID_MARKER_START("multMatColVetOPT");
multMatColVetOPT (mCol, vet, n, n, resCol);
LIKWID_MARKER_STOP("multMatColVetOPT");

LIKWID_MARKER_CLOSE;

//norma = normaMax(resRow, resPtr, n);

//norma = normaEucl(resCol, n);
```

Figura 3.3: Marcadores do LIKWID utilizados no programa matmult.

É fundamental destacar que nos dois grupos de algoritmos, apenas foram testados os algoritmos escolhidos para serem otimizados. Sendo assim, no programa `matmult`, as funções a seguir não foram executadas nos testes:

- `normaMax`;
- `normaEucl`;
- `geraMatPtr`;
- `multMatPtrVet`;

```
LIKWID_MARKER_INIT;


LIKWID_MARKER_START("gaussJacobiOPT");
verificador = gaussJacobiOPT (SL, EPS, &normaL2, &iter);
LIKWID_MARKER_STOP ("gaussJacobiOPT");

LIKWID_MARKER_START("gaussSeidelOPT");
verificador = gaussSeidelOPT (SL, EPS, &normaL2, &iter);;
LIKWID_MARKER_STOP ("gaussSeidelOPT");


LIKWID_MARKER_CLOSE;
```

Figura 3.4: Marcadores do LIKWID utilizados no programa SistemasLineares.

No programa `SistemasLineares`, por sua vez, as funções a seguir não foram executadas nos testes:

- `normaL2Residuo`;
- `eliminacaoGauss`;
- `encontraMax`;
- `normaMax`;

Devido a essa mudança nos testes realizados, foi preciso que os parâmetros de avaliação fossem novamente refeitos para as funções sem a otimização proposta. Isso permitiu diminuir o tempo de execução drasticamente de ambos os grupos de programas.

Ainda, é importante ressaltar que, apesar de ter sido feita uma separação entre dois grupos maiores de testes, ambos os grupos, para programa, são compilados com a *flag* avx, o que permite a inclusão de de *flags* de compilação, como -O3, -mavx e -march=native.

3.1 *SCRIPT*

Com o propósito de automatizar a execução dos testes, foi criado um *script* em *shell* para que sejam realizados todos os testes para determinado programa. Simplificadamente, o código-fonte do *script* pode ser separado em dois grandes blocos de código, como mostrado na Figura 3.5.

Esse *script* permite definir a ordem das matrizes que serão utilizadas para efetuar os testes, bem como o tamanho da imagem do gráfico gerado e a escala de valores no eixo das abscissas.

A ideia central desse algoritmo de teste é direcionar a impressão do resultado dos testes, seja da medição do tempo de execução, seja dos outros grupos de teste, para um arquivo

```
#PARÂMETRO $1 = CORE
#PARÂMETRO $2 = PROGRAMA
LIKWID_CMD="likwid-perfctr -C $1"

#Ordem das matrizes que serão efetuados os testes para ambos os métodos:
elements=(32 50 64 100 128 200 256 300 400 512 1000 1024 2000 2048 3000 4000 4096 5000 10000)

#Escala de valores em x
x_range="[10:10000]"

#Resolução de saída dos gráficos gerados pelo gnuplot.
image_size="1600,900" #1600x900

#TESTE DE PERFORMANCE AUTOMATIZADO PARA O PROGRAMA: matmult.
if [[ "$2" == "matmult" ]]; then
    #execução dos testes de geração dos gráficos
    # ...

#TESTE DE PERFORMANCE AUTOMATIZADO PARA O PROGRAMA: SistemasLineares
elif [[ "$2" == "SistemasLineares" ]]; then
    #execução dos testes de geração dos gráficos
    # ...

else
    echo "Os parâmetros nessários são: <CORE DA MÁQUINA A SER UTILIZADO> <PROGRAMA>."
    echo "As opções de programas são: matmult ou SistemasLineares."
    echo "Por favor, entre com os parâmetros corretos para que o script possa efetuar os testes."
fi
```

Figura 3.5: Parte do *script* usado na automação dos testes.

temporário. Esse arquivo temporário será utilizado pelo programa do `gnuplot` para a construção do gráfico de um determinado teste. Em alguns casos foi preciso formatar a saída das funções utilizando os comandos `grep`, `awk` e `tr` para construir uma tabela no arquivo temporário, em que a primeira coluna da tabela representa a ordem da matriz e as demais os resultados obtidos no teste realizado pelas diferentes funções, como exemplificado na Figura 3.6.

É importante ressaltar que esse *script* leva em conta que as variáveis de ambiente do GNU Linux já estão configuradas corretamente para a execução do programa do LIKWID e que a frequência do *clock* do processador já está configurado e fixado no máximo. É possível configurar esses parâmetros com os comandos da Figura 3.7.

3.2 *FORMA DE EXECUÇÃO DOS TESTES*

Uma vez que o *script* é capaz de realizar todos os testes apresentados anteriormente de forma automatizada, é necessário apenas fornecer dois parâmetros para o código, como: `./execute.sh <CORE> <PROGRAMA>`, onde

- `CORE`: define o núcleo de processamento a ser utilizado no teste;
- `PROGRAMA`: define o conjunto de algoritmos a ser testado, que pode ser `matmult` ou `SistemasLineares`.

É importante ressaltar que, em consequência do processo de automação de geração dos resultados, não é possível obter os gráficos de testes individualmente para um determinado grupo sem antes comentar partes do código para mudar o fluxo de execução. Sendo assim, uma vez que é lançado o algoritmo, os quatro testes apresentados anteriormente serão executados para todas as funções presentes no programa que foi passado como parâmetro para o *script*.

```
    for size in ${elements[@]} ;
    do
        ${LIKWID_CMD} -f -g ${group} -m $PROGRAM -n $size > temp.tmp

        printf "$((size))"
        printf "$(grep "$key" temp.tmp | awk -F"|" '{print $3}' | tr "\n" " ")\n"

    done > $group.tmp

    gnuplot <<- EOF
    reset
    set terminal png size ${image_size} enhanced font 'Verdana,12'
    set style data linespoints
    set style fill solid 2.00 border 0

    set style line 1 lc rgb 'orange' lt 1 lw 2 pt 13 ps 1.0
    set style line 2 lc rgb 'red' lt 1 lw 2 pt 7 ps 1.0

    set key font ",10"
    set key horizontal
    set key spacing 3
    set key samplen 3
    set key width 2

    set title 'Medição de desempenho para $group-multMatPtrVet' font ",16"

    set xrange ${x_range}
    set logscale x

    set xlabel "tamanho da matriz (doubles)"
    set xlabel font ",13"
    set ylabel "$magnitude"
    set ylabel font ",13"

    set output "$group-multMatPtrVet.png"
    plot '$group.tmp' using 1:2 with linespoints ls 1 title "DP", \
    '$group.tmp' using 1:3 with linespoints ls 2 title "AVX DP"
EOF
```

Figura 3.6: Formatação das saídas geradas pelos marcadores para a construção da tabela.

```
export PATH=/home/soft/likwid/bin:/home/soft/likwid/sbin:$PATH

export LD_LIBRARY_PATH=/home/soft/likwid/lib:$LD_LIBRARY_PATH

echo "performance" > /sys/devices/system/cpu/cpufreq/policy3/scaling_governor
```

Figura 3.7: Alteração de variáveis de ambiente e aumento do *clock* da máquina.

4 RESULTADOS OBTIDOS

Este capítulo apresenta os resultados obtidos dos quatro testes realizados em cada uma das quatro funções apresentadas anteriormente, totalizando um total de dezesseis gráficos gerados. Esses gráficos foram agrupados em uma única imagem para facilitar a visualização de todos os resultados para uma determinada função. Ainda, como os parâmetros de testes foram alterados, esses quatro testes também foram aplicados às funções sem otimização.

Após a realização de vários experimentos, nas mesmas condições descritas no Capítulo 2, verificou-se que o tempo médio para a conclusão da análise de desempenho para o programa `matmult` e para o programa `SistemasLinearesfoi` de aproximadamente 3 minutos. Isso levou a uma queda drástica do tempo de execução quando comprado aos 35 minutos descritos no relatório anterior.

Com o objetivo de comparar melhor os resultados, será feita uma análise por tópicos da seguinte maneira:

- `TEMPO`: representa o tempo gasto, em milissegundos, para cada uma das diferentes ordem de matrizes a que foram submetidos os testes;
- `L3 bandwidth`: representa a quantidade máxima de dados que podem ser transferidos por segundo de uma fonte (memória principal ou *cache*) para o destino (registradores);
- `L2 cache miss ratio`: representa a taxa com que o processador foi buscar um dado na memória principal pois o dado não estava contido na *cache* L2;
- `FLOPS_DP`: representa a quantidade de operações em ponto flutuante de precisão dupla que são realizados, utilizando ou não instruções do tipo AVX[1]

4.1 *MATMULT*

O resultado dos testes para o grupo do `matmult` geraram os seguinte gráficos das Figuras 4.3 e 4.4, que foram agrupados. Ainda, de forma a estabelecer um comparativo entre os dois algoritmos, foram gerados os seguintes gráficos das Figuras 4.1 e 4.2, ambas sem as otimizações propostas.

Na computação, esses tipos de algoritmos são maneiras clássicas de perceber a importância de conhecer o *layout* com que os dados estão armazenados na memória em diferentes tipos de linguagens. Basicamente, há duas maneiras para armazenas *arrays* multidimensionais em memória de forma contígua, como:

[1]São extensões da arquitetura do conjunto de instruções x86 para microprocessadores da Intel e da AMD. Essa extensão é capaz de suportar dezesseis registradores (8 números flutuante de precisão simples de 32 bits ou 4 números de ponto flutuante de precisão dupla de 64 bits) Lomont (2011).

- `column major order`: os dados próximos no *array* estão organizados em colunas. Um exemplo de linguagem de programação que utiliza esse tipo de organização é o Fortran.

- `row major order`: os dados próximos no *array* estão organizados em linhas. Um exemplo de linguagem de programação que utiliza esse tipo de organização é o C/C++.

Após a observação não só dos algoritmos implementados por esse programa, mas também dos resultados obtidos da análise de *performance* realizados, podemos inferir alguns pontos quando comparando os resultados obtidos:

- `TEMPO`: o tempo de execução do algoritmo do multMatRowVet caiu drasticamente quando comparado com o resultado anterior (cerca de 10 x menos). O algoritmo do multMatColVet, por sua vez, não obteve um resultado satisfatório na redução do tempo de execução.

- `L3 bandwidth`: o algoritmo do multMatRowVet teve sua largura de banda aumentada drasticamente em comparação ao resultado anterior (de 24.000 até cerca de 40.000 MBytes/s). Isso se deve ao fato das otimizações feitas permitirem um alto fluxo de dados pelo *cache* L3. Já o algoritmo do multMatColVet, o resultado para esse parâmetro foi muito similar ao teste realizado anteriormente sem otimização.

- `L2 cache miss ratio`: nesse parâmetro, o algoritmo do multMatRowVet apresentou uma piora nos resultados, quando comparado à versão sem otimização, na medida em que a ordem de grande das matrizes crescia. O algoritmo do multMatColVet, por sua vez, manteve quase que semelhante ao resultado anterior. No entanto, em algumas ordem de matrizes, a taxa de *cache miss* foi maior quando comparado ao algoritmo anterior.

- `FLOPS_DP`: nesse parâmetro, o algoritmo do multMatRowVet teve um desempenho muito satisfatório. Esse algoritmo conseguiu manter uma taxa elevada de operações em precisão dupla realizados pelo processador (uma melhoria de quase o dobro, quando comparado ao resultado anterior). Ainda, esse algoritmo conseguiu utilizar os registradores que utilizam o conjunto AVX de instruções, o que representa um ganho considerável na aplicação. Já algoritmo do multMatColVet, apresentou uma elevação da utilização de operações em ponto flutuante de precisão dupla realizados pelo processador e até chegou a utilizar o conjunto AVX em algumas ordens de matrizes, porém, essa utilização não se manteve.

4.2 *SISTEMASLINEARES*

O resultado dos testes para o grupo do `SistemasLineares` geraram os seguinte gráficos das Figuras 4.7 e 4.8, que foram agrupados. Ainda, de forma a estabelecer um

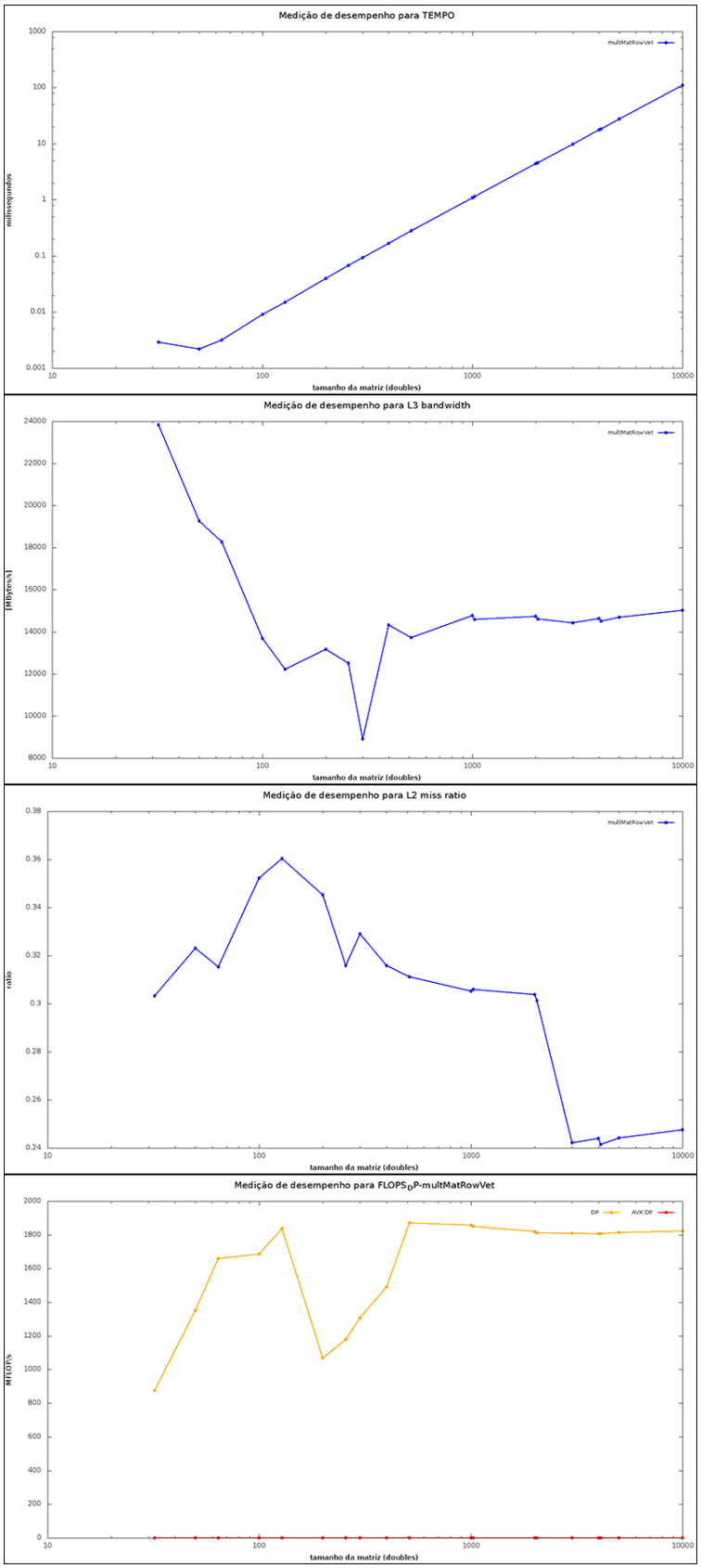

Figura 4.1: Resultados obtidos para a função multMatRowVet, SEM OTIMIZAÇÃO.

comparativo entre os dois algoritmos, foram gerados os seguintes gráficos das Figuras 4.5 e 4.6, ambas sem as otimizações propostas.

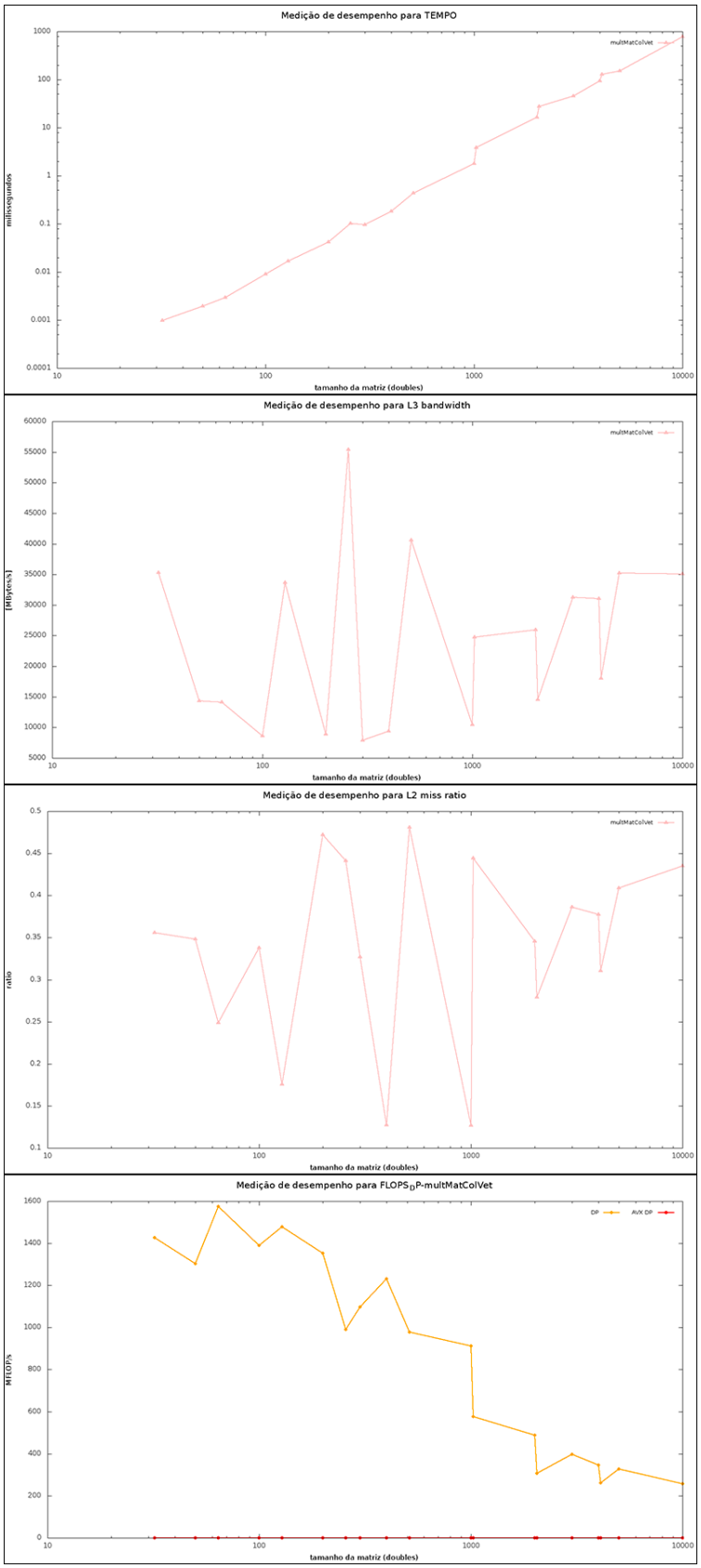

Figura 4.2: Resultados obtidos para a função multMatColVet, SEM OTIMIZAÇÃO.

Na computação científica, esses tipos de algoritmos são muito utilizados no cálculo de Soluções de Sistemas de Equações Lineares. Assim, quanto melhor for o desempenho de um algoritmo quando comparado a um outro, melhor ele será explorado pelos programadores.

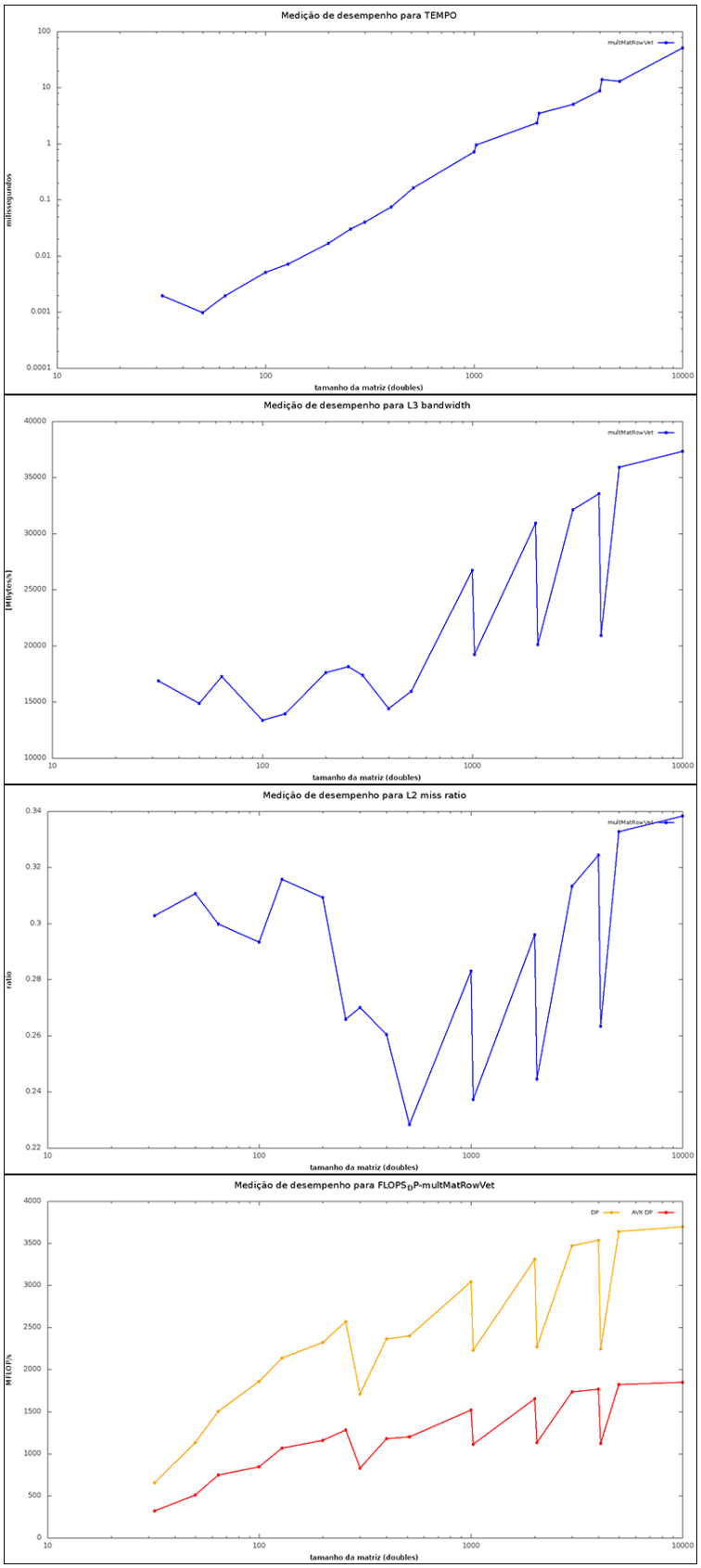

Figura 4.3: Resultados obtidos para a função multMatRowVet, COM OTIMIZAÇÃO.

Após a observação dos algoritmos implementados por esse programa e dos resultados obtidos da análise de *performance* realizados, podemos inferir alguns pontos:

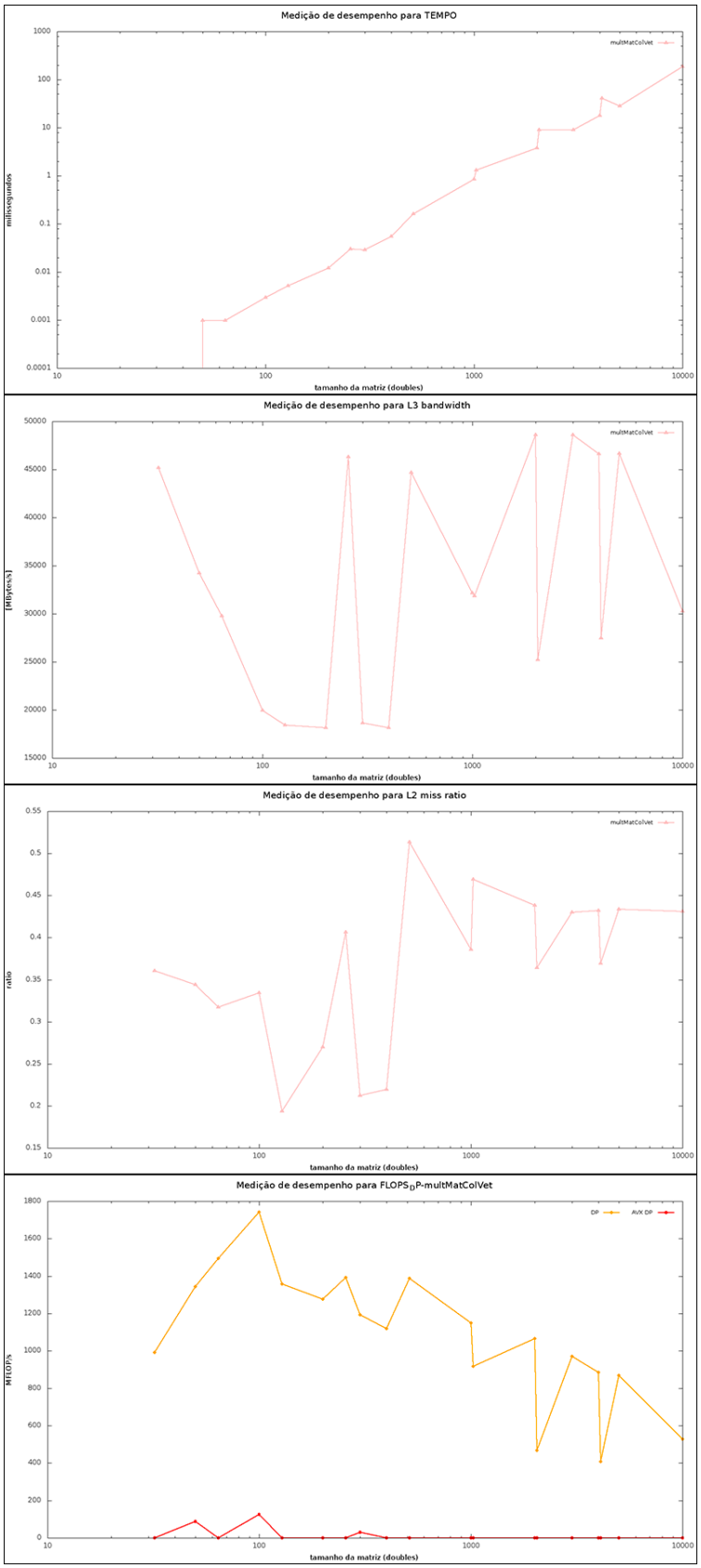

Figura 4.4: Resultados obtidos para a função multMatColVet, COM OTIMIZAÇÃO.

- `TEMPO`: o tempo de execução do algoritmo do gaussJacobi e do gaussSeidel caíram drasticamente quando comparado com o resultado anterior de ambos (cerca de 10 x menos).

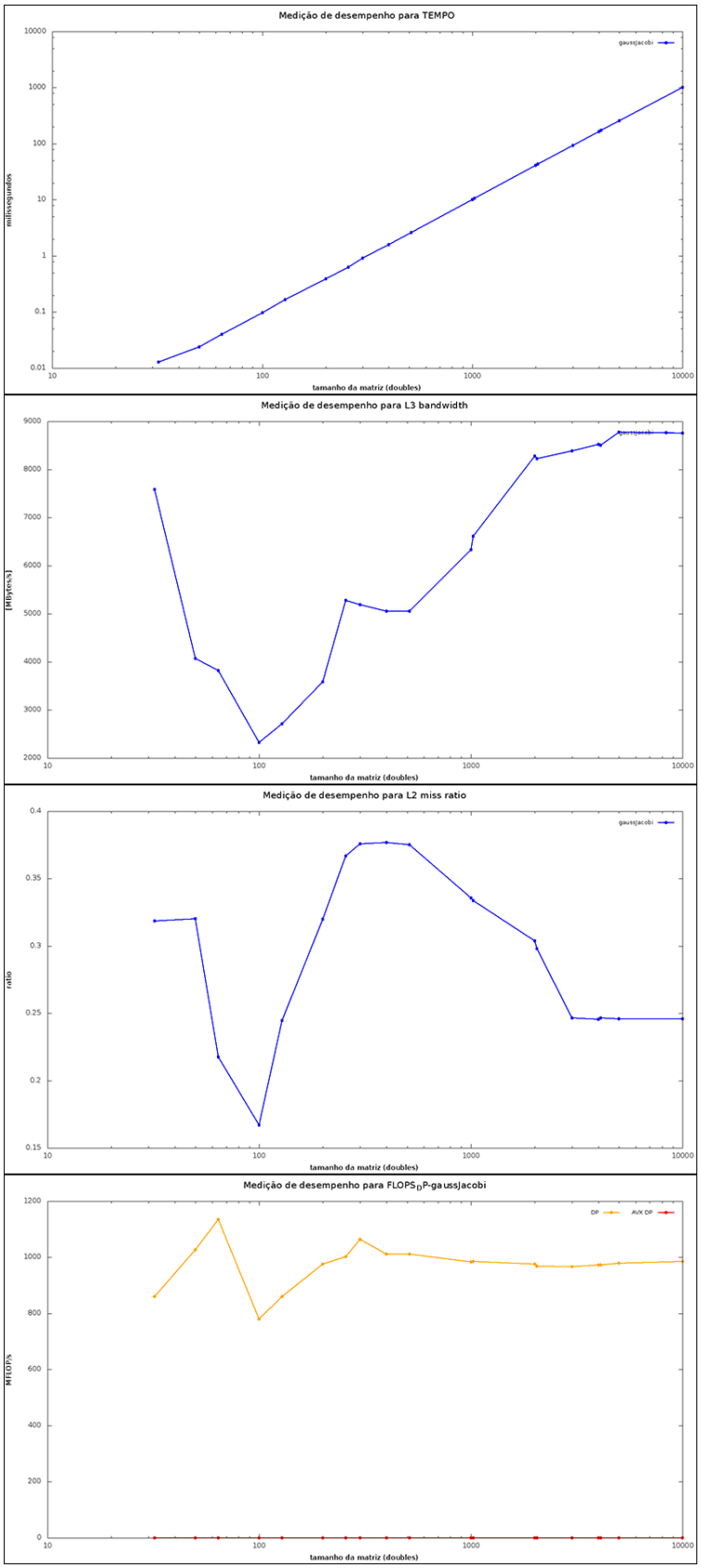

Figura 4.5: Resultados obtidos para a função gaussJacobi, SEM OTIMIZAÇÃO.

- `L3 bandwidth`: o algoritmo do gaussJacobi teve sua largura de banda no mesmo nível quando comparado ao resultado anterior. A medida em que a ordem da matriz

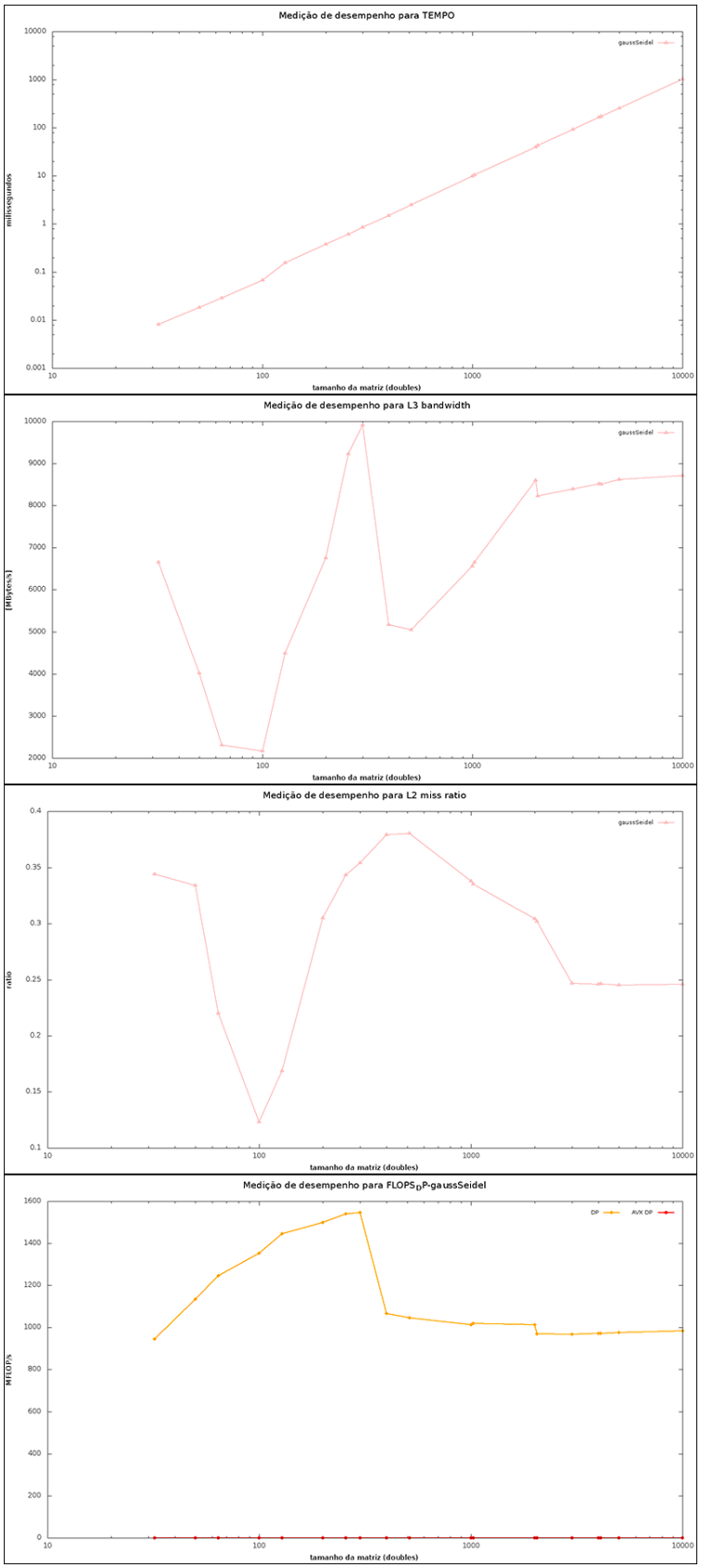

Figura 4.6: Resultados obtidos para a função gaussSeidel, SEM OTIMIZAÇÃO.

crescia, essa largura de banda era diminuída. Já o algoritmo do gaussSeidel, o limite

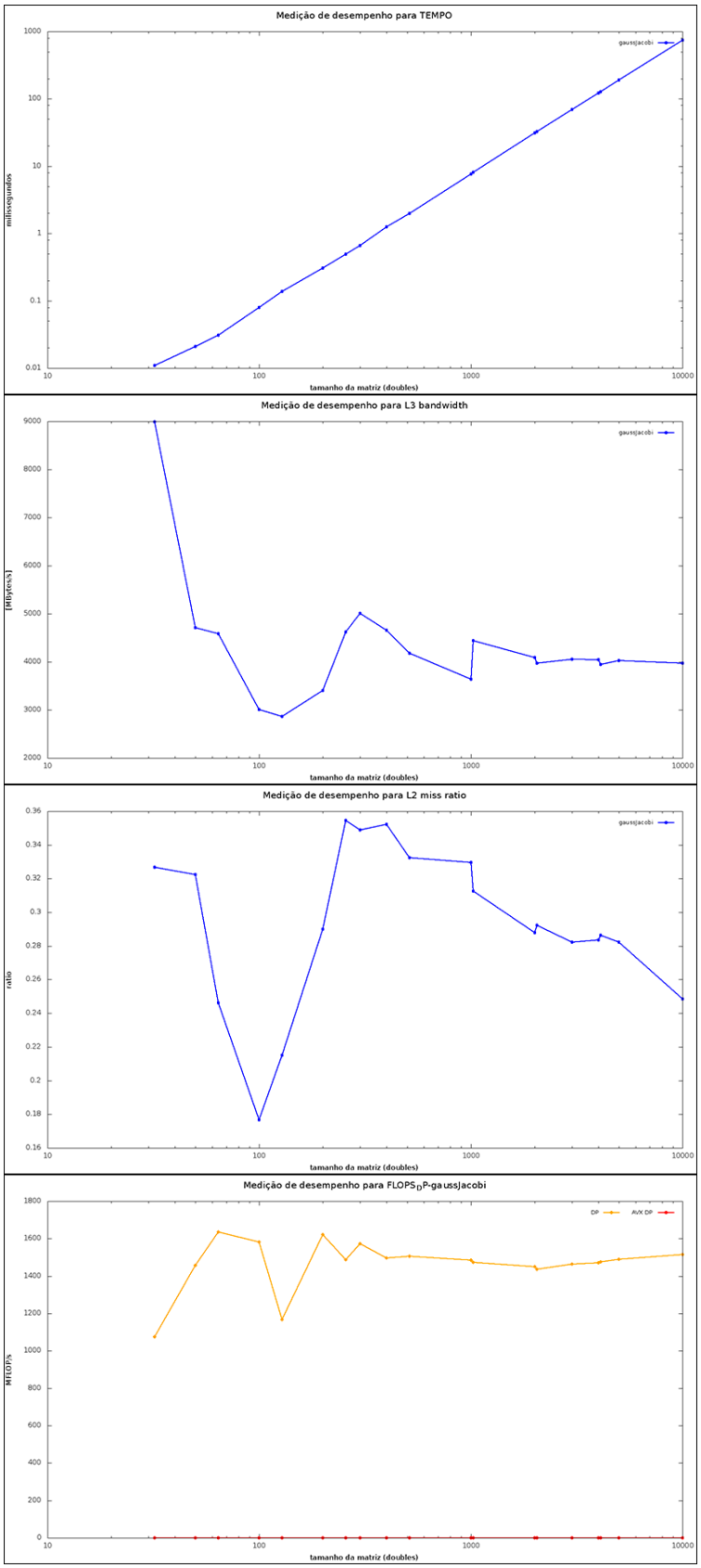

Figura 4.7: Resultados obtidos para a função gaussJacobi, COM OTIMIZAÇÃO.

da largura de banda foi levemente aumentado, porém, assim como o algoritmo de

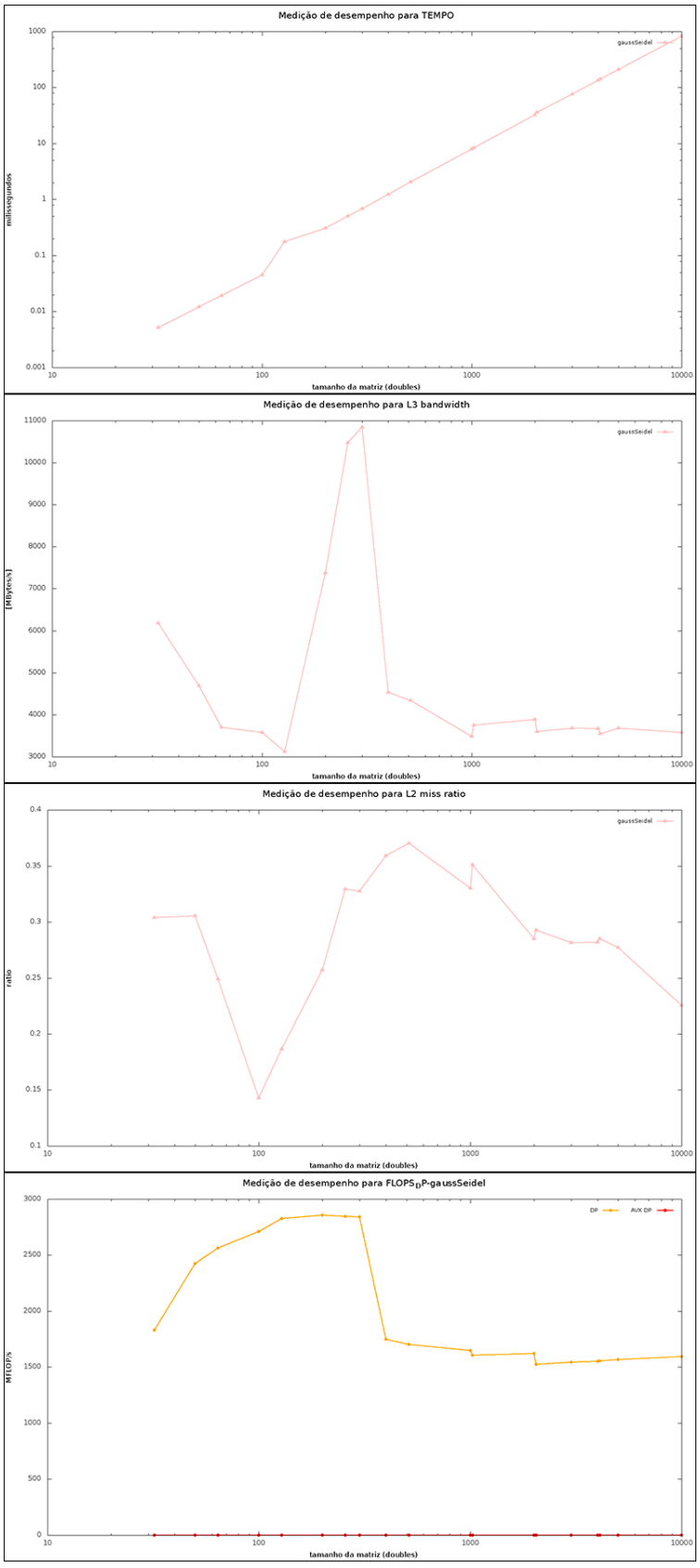

Figura 4.8: Resultados obtidos para a função gaussSeidel, COM OTIMIZAÇÃO.

gaussJacobi, a medida em que a ordem da matriz crescia, essa largura de banda era diminuída.

- `L2 cache miss ratio`: nesse parâmetro, o algoritmo do gaussJacobi obteve uma leve melhora nos seus índices, enquanto que o algoritmo do gaussSeidel manteve praticamente idêntico ao resultado anterior do algoritmo sem otimização.

- `FLOPS_DP`: nesse parâmetro, ambos os algoritmos conseguiram aumentar a taxa de operações em ponto flutuante de precisão dupla realizados pelo processador. No entanto, não foi obtido êxito em utilizar os registradores que suportam instruções AVX pelo processador.

5 CONCLUSÃO

Após o processo de análise de todos os gráficos gerados dos algoritmos que foram testados pelo *script* foi constatado que os algoritmos, com relação ao tempo de execução, obtiveram um ganho considerável. No entanto, a maioria dos algoritmos não conseguiram utilizar o conjunto de instruções AVX, o que os tornam um pouco ineficientes quando comparados com algoritmos semelhantes que utilizam esse conjunto.

REFERÊNCIAS

Lomont, C. (2011). Introduction to intel® advanced vector extensions. https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions. Acessado em 02/06/2019.

Prado, S. (2017). Otimização de código em linguagem c – parte 1. https://sergioprado.org/otimizacao-de-codigo-em-linguagem-c-parte-1/. Acessado em 17/06/2019.